下载APP

19 | Python编译器（三）：运行时机制

2020-07-15 宫文学

编译原理实战课 进入课程 >

讲述：宫文学
时长 26:01 大小 23.84M

你好，我是宫文学。

在前面两讲中，我们已经分析了 Python 从开始编译到生成字节码的机制。但是，我们对 Python 只是了解了一半，还有很多问题需要解答。比如：**Python 字节码是如何运行的呢？它是如何管理程序所用到的数据的？它的类型体系是如何设计的，有什么特点？**等等。

所以今天这一讲，我们就来讨论一下 Python 的运行时机制。其中的**核心，是 Python 对象机制的设计**。

我们先来研究一下字节码的运行机制。你会发现，它跟 Python 的对象机制密切相关。

理解字节码的执行过程

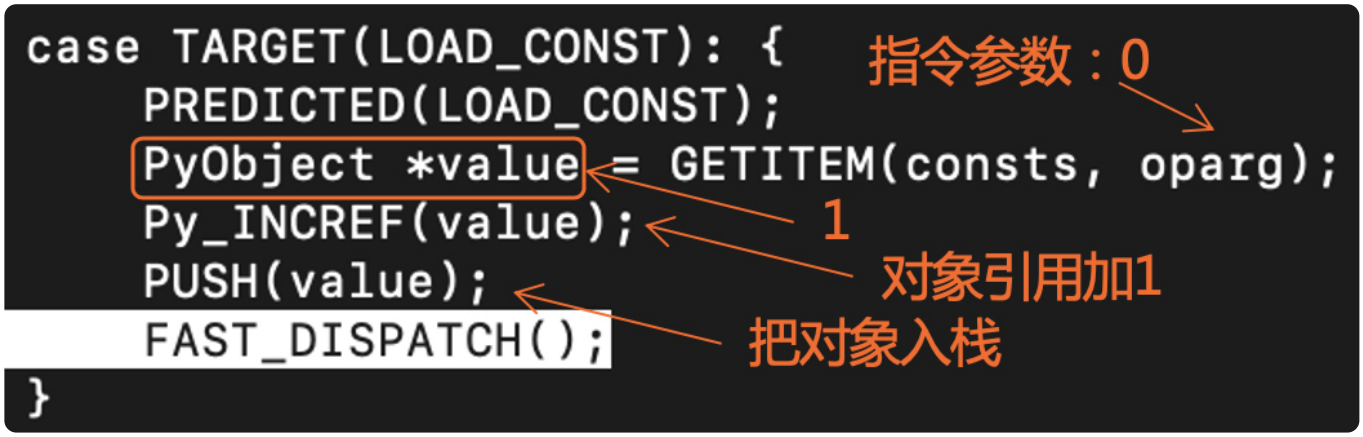
我们用 GDB 跟踪执行一个简单的示例程序，它只有一行：“a=1”。

这行代码对应的字节码如下。其中，前两行指令实现了“a=1”的功能（后两行是根据 Python 的规定，在执行完一个模块之后，缺省返回一个 None 值）。

```
0 LOAD_CONST               0 (1)
2 STORE_NAME               0 (a)
4 LOAD_CONST               1 (None)
6 RETURN_VALUE
```

你需要在 **_PyEval_EvalFrameDefault() 函数**这里设置一个断点，在这里实际解释指令并执行。

首先是执行第一行指令，LOAD_CONST。



你会看到，解释器做了三件事情：

1. 从常数表里取出 0 号常数。你知道，编译完毕以后会形成 PyCodeObject，而在这个对象里会记录所有的常量、符号名称、本地变量等信息。常量 1 就是从它的常量表中取出来的。
2. 把对象引用值加 1。对象引用跟垃圾收集机制相关。
3. 把这个常数对象入栈。

从这第一行指令的执行过程，你能得到什么信息呢？

第一个信息，常量 1 在 Python 内部，它是一个对象。你可以在 GDB 里显示这个对象的信息：该对象的类型是 PyLong_Type 型，这是 Python 的整型在内部的实现。

另外，该对象的引用数是 126 个，说明这个常量对象其实是被共享的，LOAD_CONST 指令会让它的引用数加 1。我们用的常数是 1，这个值在 Python 内部也是会经常被用到，所以引用数会这么高。你可以试着选个不那么常见的常数，看看它的引用数是多少，都是在哪里被引用的。

进一步，我们会发现，往栈里放的数据，其实是个对象指针，而不像 Java 的栈机那样，是放了个整数。

总结上述信息，我其实可以告诉你一个结论：**在 Python 里，程序中的任何符号都是对象，包括整数、浮点数这些基础数据，或者是自定义的类，或者是函数，它们都是对象。**在栈机里处理的，是这些对象的引用。

我们再继续往下分析一条指令，也就是 STORE_NAME 指令，来加深一下对 Python 运行机制的理解。

```
case TARGET(STORE_NAME): {
    PyObject *name = GETITEM(names, oparg);
    PyObject *v = POP();
    PyObject *ns = f->f_locals;
    int err;
    if (ns == NULL) {
        _PyErr_Format(tstate, PyExc_SystemError,
                      "no locals found when storing %R", name);
        Py_DECREF(v);
        goto error;
    }
    if (PyDict_CheckExact(ns))
        err = PyDict_SetItem(ns, name, v);
    else
        err = PyObject_SetItem(ns, name, v);
    Py_DECREF(v);
    if (err != 0)
        goto error;
    DISPATCH();
}
```


执行 STORE_NAME 指令时，解释器做了 5 件事情：

1. 根据指令的参数，从名称表里取出变量名称。这个名称表也是来自于 PyCodeObject。前面我刚说过了，Python 程序中的一切都是对象，那么 name 也是对象。你可以查看它的类型，是 PyUnicode_Type，也就是 Unicode 的字符串。
2. 从栈顶弹出上一步存进去的常量对象。
3. 获取保存了所有本地变量的字典，这也是来自 PyCodeObject。
4. 在字典里，设置 a 的值为该常量。如果你深入跟踪其执行过程，你会发现在存入字典的时候，name 对象和 v 对象的引用都会加 1。这也是可以理解的，因为它们一个作为 key，一个作为 value，都要被字典所引用。
5. 减少常量对象的引用计数。意思是栈机本身不再引用该常量。

好了，通过详细解读这两条指令的执行过程，我相信你对 Python 的运行机制摸到一点头绪了，但可能还是会提出很多问题来，比如说：

既然栈里的操作数都是对象指针，那么如何做加减乘除等算术运算？

如果函数也是对象，那么执行函数的过程又是怎样的？

……

别着急，我在后面会带你探究清楚这些问题。不过在此之前，我们有必要先加深一下对 Python 对象的了解。

Python 对象的设计

Python 的对象定义在 object.h 中。阅读文件头部的注释和对各类数据结构的定义，你就可以理解 Python 对象的设计思路。

首先是 PyObject 和 PyVarObject 两个基础的数据结构，它们分别表示定长的数据和变长的数据。

复制代码

```
typedef struct _object {           //定长对象
    Py_ssize_t ob_refcnt;          //对象引用计数
    struct _typeobject *ob_type;   //对象类型
} PyObject;

typedef struct {          //变长对象
    PyObject ob_base;
    Py_ssize_t ob_size;   //变长部分的项目数量，在申请内存时有确定的值，不再变
} PyVarObject;
```

PyObject 是最基础的结构，所有的对象在 Python 内部都表示为一个 PyObject 指针。它里面只包含两个成员：对象引用计数（ob_refcnt）和对象类型（ob_type），你在用 GDB 跟踪执行时也见过它们。可能你会问，**为什么只有这两个成员呢？对象的数据（比如一个整数）保存在哪里？**

实际上，任何对象都会在一开头包含 PyObject，其他数据都跟在 PyObject 的后面。比如说，Python3 的整数的设计是一个变长对象，会用一到多个 32 位的段，来表示任意位数的整数：

复制代码

```
#define PyObject_VAR_HEAD      PyVarObject ob_base;
struct _longobject {
    PyObject_VAR_HEAD       //PyVarObject
    digit ob_digit[1];      //数字段的第一个元素
};
```

```
6 typedef struct _longobject PyLongObject;     //整型
```

它在内存中的布局是这样的：

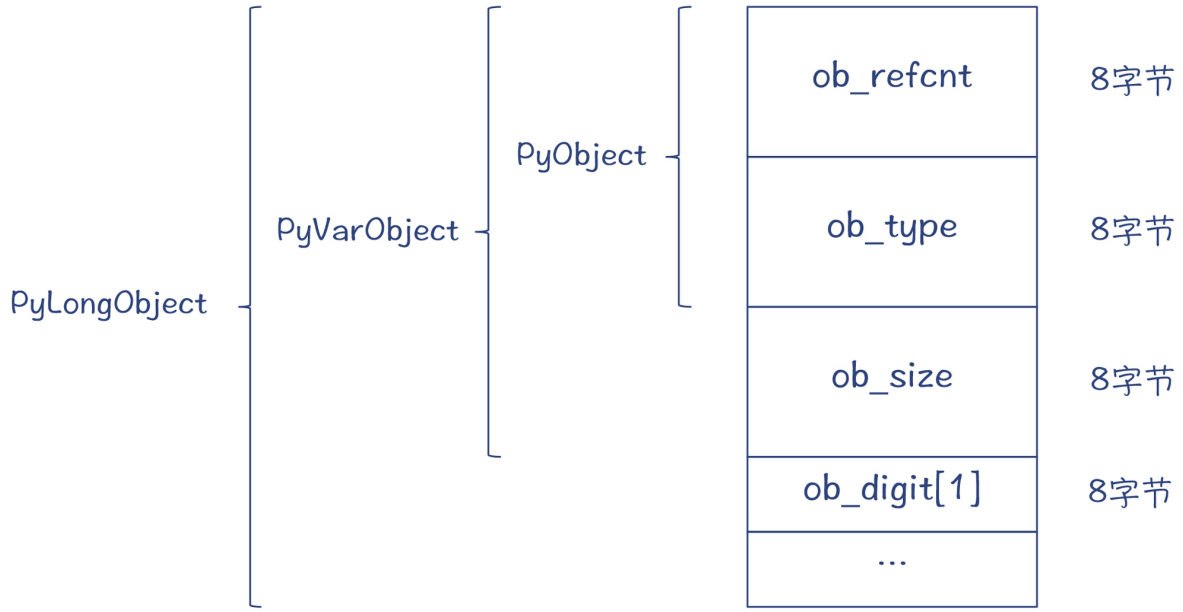


图 1：Python3 的整数对象的内存布局

所以你会看出，**PyObject*、PyVarObject*和PyLongObject*指向的内存地址是相同的**。你可以根据 ob_type 的类型，把PyObject*强制转换成PyLongObject*。

实际上，像 C++ 这样的面向对象语言的内存布局也是如此，父类的成员变量在最前面，子类的成员变量在后面，父类和子类的指针之间可以强制转换。懂得了这个原理，我们用 C 语言照样可以模拟出面向对象的继承结构出来。

你可以注意到，我在图 1 中标出了每个字段所占内存的大小，总共是 28 个字节（这是 64 位 macOS 下的数值，如果是另外的环境，比如 32 位环境，数值会有所不同）。

你可以用 sys.getsizeof() 函数，来测量对象占据的内存空间。

复制代码

```
1 >>> a = 10
2 >>> import sys
3 >>> sys.getsizeof(a)
4 28       #ob_size = 1
5 >>> a = 1234567890
```

```
>>> sys.getsizeof(a)
32        #ob_size = 2，支持更大的整数
```

到这里，我们总结一下 Python 对象设计的三个特点：

1. 基于堆

Python 对象全部都是在堆里申请的，没有静态申请和在栈里申请的。这跟 C、C++ 和 Java 这样的静态类型的语言很不一样。

C 的结构体和 C++ 的对象都既可以在栈里，也可以在堆里；Java 也是一样，除了原生数据类型可以在栈里申请，未逃逸的 Java 对象的内存也可以在栈里管理，我们在讲Java 的 JIT 编译器的时候已经讲过了。

2. 基于引用计数的垃圾收集机制

每个 Python 对象会保存一个引用计数。也就是说，Python 的垃圾收集机制是基于引用计数的。

它的优点是可以实现增量收集，只要引用计数为零就回收，避免出现周期性的暂停；缺点是需要解决循环引用问题，并且要经常修改引用计数（比如在每次赋值和变量超出作用域的时候），开销有点大。

3. 唯一 ID

每个 Python 对象都有一个唯一 ID，它们在生存期内是不变的。用 id() 函数就可以获得对象的 ID。根据 Python 的文档，这个 ID 实际就是对象的内存地址。所以，实际上，你不需要在对象里用一个单独的字段来记录对象 ID。这同时也说明，Python 对象的地址在整个生命周期内是不会变的，这也符合基于引用计数的垃圾收集算法。对比一下，如果采用“停止和拷贝”的算法，对象在内存中会被移动，地址会发生变化。所以你能看出，ID 的算法与垃圾收集算法是环环相扣的。

复制代码

```
>>> a = 10
>>> id(a)
```

```
3 140230830057200
```

接下来，我们看看 ob_type 这个字段，它指向的是对象的类型。以这个字段为线索，我们就可以牵出 Python 的整个类型系统的设计。

Python 的类型系统

Python 是动态类型的语言。它的类型系统的设计相当精巧，Python 语言的很多优点，都来自于它的类型系统。我们来看一下。

首先，Python 里每个 PyObject 对象都有一个类型信息。保存类型信息的数据结构是 PyTypeObject（定义在 Include/cpython/object.h 中）。PyTypeObject 本身也是一个 PyObject，只不过这个对象是用于记录类型信息的而已。它是一个挺大的结构体，包含了对一个类型的各种描述信息，也包含了一些函数的指针，这些函数就是对该类型可以做的操作。可以说，只要你对这个结构体的每个字段的作用都了解清楚了，那么你对 Python 的类型体系也就了解透彻了。

复制代码

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name;  /* 用于打印的名称格式是"<模块>.<名称>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* 用于申请内存 */

    /* 后面还有很多字段，比如用于支持数值计算、序列、映射等操作的函数，用于描述属性、子类型、
    ...
} PyTypeObject
```

因为这个结构非常重要，所以我把一些有代表性的字段整理了一下，你可以重点关注它们：

PyTypeObject Slot（插槽）	类型	特殊方法/属性	
tp_name	const char *	__name__	类型名称
tp_basicsize	Py_ssize_t		对象总的大小=tp_basicsize + tp_itemsize * ob_size
tp_itemsize	Py_ssize_t		
tp_dealloc	destructor		释放对象内存时调用
tp_repr	reprfunc	__repr__	用于描述对象的字符串
			结构体，里面保存了与数字运算

tp_as_number	PyNumberMethods *		有关的函数，比如add、subtract、multiply、divmod、or、xor、and等几十个函数
tp_as_sequence	PySequenceMethods *		与序列有关的函数，比如获得长度、连接、重复等
tp_as_mapping	PyMappingMethods *		与映射有关的函数
tp_hash	hashfunc	__hash__	哈希函数
tp_call	ternaryfunc	__call__	对象调用 任何带有tp_call函数的类型，意味着该类型的实例是可被调用的(Callable)
tp_str	reprfunc	__str__	转字符串，实现str()函数
tp_getattro	getattrofunc	__getattribute__, __getattr__	获取属性
tp_setattro	setattrofunc	__setattr__, __delattr__	设置属性
tp_doc	const char *	__doc__	文档
tp_traverse	traverseproc		与垃圾收集有关的函数
tp_clear	inquiry		
tp_richcompare	richcmpfunc	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	支持关系运算的函数。一个函数可以支持多个操作符
tp_iter	getiterfunc	__iter__	枚举器
tp_iternext	iternextfunc	__next__	
tp_methods	PyMethodDef []		常规方法描述信息 数组中的每个方法，在__dict__里都有一个实际函数
tp_members	PyMemberDef []		常规的数据成员的描述信息
tp_getset	PyGetSetDef []		计算属性的getter和setter函数
tp_base	PyTypeObject *	__base__	基类
tp_dict	PyObject *	__dict__	所有属性和方法的字典 这个是最常用到的一个函数了
tp_init	initproc	__init__	对象初始化函数
tp_alloc	allocfunc		
tp_new	newfunc	__new__	创建新对象
tp_free	freefunc		
tp_is_gc	inquiry		
<tp_bases>	PyObject *	__bases__	多重基类的列表
<tp_mro>	PyObject *	__mro__	方法查找顺序
[tp_subclasses]	PyObject *	__subclasses__	针对子类型的弱引用
tp_finalize	destructor	__del__	析构函数

你会看到，这个结构里的很多部分都是一个函数插槽（Slot），你可以往插槽里保存一些函数指针，用来实现各种标准操作，比如对象生命周期管理、转成字符串、获取哈希值等。

在上面的表格中，你还会看到像“__init__”这样的方法，它的两边都是有两个下划线的，也就是“double underscore”，简称 dunder 方法，也叫做“**魔术方法**”。在用 Python 编写自定义的类的时候，你可以实现这些魔术方法，它们就会被缺省的tp_*函数所调用，比如，“__init__”会被缺省的tp_init函数调用，完成类的初始化工作。

现在我们拿整型对象来举个例子，一起来看看它的 PyTypeObject 是什么样子。

对于整型对象来说，它的 ob_type 会指向一个 PyLong_Type 对象。这个对象是在 longobject.c 中初始化的，它是 PyTypeObject 的一个实例。从中你会看到一些信息：类型名称是“int”，转字符串的函数是long_to_decimal_string，此外还有比较函数、方法描述、属性描述、构建和析构函数等。

```
PyTypeObject PyLong_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)        类型的类型，也就是元类
    "int",                                        /* tp_name */
    offsetof(PyLongObject, ob_digit),             /* tp_basicsize */
    sizeof(digit),                                /* tp_itemsize */
    0,                                            /* tp_dealloc */
    0,                                            /* tp_vectorcall_offset */
    0,                                            /* tp_getattr */
    0,                                            /* tp_setattr */
    0,                                            /* tp_as_async */
    long_to_decimal_string,                       /* tp_repr */
    &long_as_number,                              /* tp_as_number */
    0,                                            /* tp_as_sequence */
    0,                                            /* tp_as_mapping */
    (hashfunc)long_hash,                          /* tp_hash */
    0,                                            /* tp_call */
    0,                                            /* tp_str */
    PyObject_GenericGetAttr,                      /* tp_getattro */
    0,                                            /* tp_setattro */
    0,                                            /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
        Py_TPFLAGS_LONG_SUBCLASS,                 /* tp_flags */
    long_doc,                                     /* tp_doc */
    0,                                            /* tp_traverse */
    0,                                            /* tp_clear */
    long_richcompare,                             /* tp_richcompare */
    0,                                            /* tp_weaklistoffset */
    0,                                            /* tp_iter */
    0,                                            /* tp_iternext */
    long_methods,                                 /* tp_methods */
    0,                                            /* tp_members */
    long_getset,                                  /* tp_getset */
    0,                                            /* tp_base */
    0,                                            /* tp_dict */
    0,                                            /* tp_descr_get */
    0,                                            /* tp_descr_set */
    0,                                            /* tp_dictoffset */
    0,                                            /* tp_init */
    0,                                            /* tp_alloc */
    long_new,                                     /* tp_new */
    PyObject_Del,                                 /* tp_free */
};
```

一个结构体，里面有各种数值运算函数的指针

一个函数，能支持各种比较运算

这里后面会被设置成PyBaseObject_Type

我们运行 type() 函数，可以获得一个对象的类型名称，这个名称就来自 PyTypeObject 的 tp_name。

复制代码

```
>>> a = 10
>>> type(a)
<type 'int'>
```

我们用 dir() 函数，可以从 PyTypeObject 中查询出一个对象所支持的所有属性和方法。比如，下面是查询一个整型对象获得的结果：

```
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '
__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__'
, '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__init_sub
class__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__
', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '_
_reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '
__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxo
r__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__tru
nc__', '__xor__', 'as_integer_ratio', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'ima
g', 'numerator', 'real', 'to_bytes']
>>>
```

好，我们刚才讲了整型，它对应的 PyTypeObject 的实例是 PyLong_Type。Python 里其实还有其他一些内置的类型，它们分别都对应了一个 PyTypeObject 的实例。你可以参考一下这个表格。

类型	对象	类型对象	举例	说明
int	PyLongObject	PyLong_Type	a = 1	整型
float	PyFloatObject	PyFloat_Type	a = 2.0	浮点型
str	PyUnicodeObject	PyUnicode_Type	a = "richard"	字符串型
list	PyListObject	PyList_Type	a = ['a', 2, 'b']	列表

dict	PyDictObject	PyDict_Type	person = {
			"name": "Richard",
			"gender": "male",
			"year": 1974
			}
				字典

code	PyCodeObject	PyCode_Type		代码
function	PyFunctionObject	PyFunction_Type	def foo():	函数型
type	PyTypeObject	PyType_Type		元类
object		PyBaseObject_Type		所有对象的基类

上面列出的这些都是 Python 的内置类型。有些内置类型跟语法是关联着的，比如说，“a = 1”就会自动创建一个整型对象，“a = [2, 'john', 3]”就会自动创建一个 List 对象。**这些内置对象都是用 C 语言实现的。**

Python 比较有优势的一点，是你可以用 C 语言，像实现内置类型一样实现自己的类型，并拥有很高的性能。

当然，如果性能不是你最重要的考虑因素，那么你也可以用 Python 来创建新的类型，也就是以 class 关键字开头的自定义类。class 编译以后，也会形成一个 PyTypeObject 对象，来代表这个类。你为这个类编写的各种属性（比如类名称）和方法，会被存到类型对象中。

好了，现在你已经初步了解了 Python 的类型系统的特征。接下来，我就带你更深入地了解一下类型对象中一些重要的函数插槽的作用，以及它们所构成的一些协议。

Python 对象的一些协议

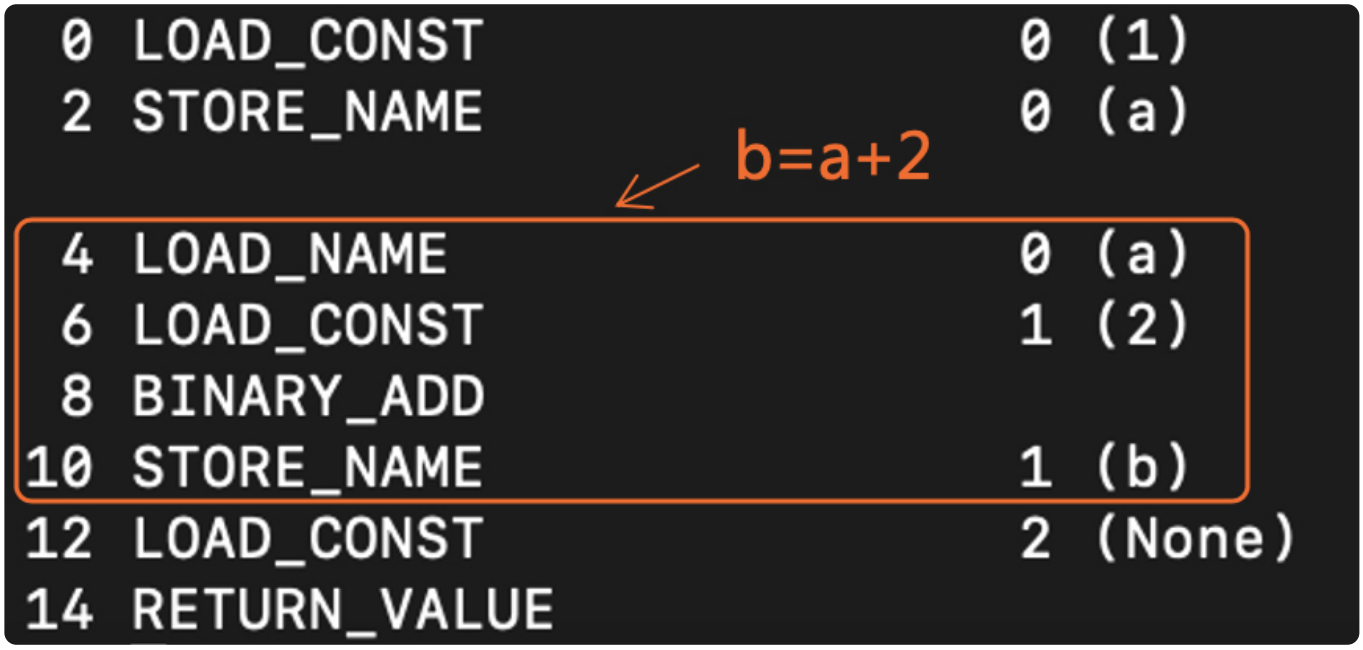
前面在研究整型对象的时候，你会发现 PyLong_Type 的 tp_as_number 字段被赋值了，这是一个结构体（PyNumberMethods），里面是很多与数值计算有关的函数指针，包括加减乘除等。这些函数指针是实现 Python 的数值计算方面的协议。任何类型，只要提供了这些函数，就可以像整型对象一样进行计算。这实际上是 Python 定义的一个针对数值计算的协议。

既然如此，我们再次用 GDB 来跟踪一下 Python 的执行过程，看看整数的加法是怎么实现的。我们的示例程序增加了一行代码，变成：

复制代码

```
1 a = 1
2 b = a + 2
```

它对应的字节码如下：



我们重点来关注 BINARY_ADD 指令的执行情况，如下图所示：

```
case TARGET(BINARY_ADD): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;
    /* NOTE(haypo): Please don't try to micro-optimize int+int on
       CPython using bytecode, it is simply worthless.
       See http://bugs.python.org/issue21955 and
       http://bugs.python.org/issue10044 for the discussion. In short,
       no patch shown any impact on a realistic benchmark, only a minor
       speedup on microbenchmarks. */
    if (PyUnicode_CheckExact(left) &&
             PyUnicode_CheckExact(right)) {
        sum = unicode_concatenate(tstate, left, right, f, next_instr);
        /* unicode_concatenate consumed the ref to left */
    }
    else {
        sum = PyNumber_Add(left, right);
        Py_DECREF(left);
    }
    Py_DECREF(right);
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
    DISPATCH();
}
```

如果加号两边都是字符串，则做连接字符串的操作。

否则，作为数字相加。

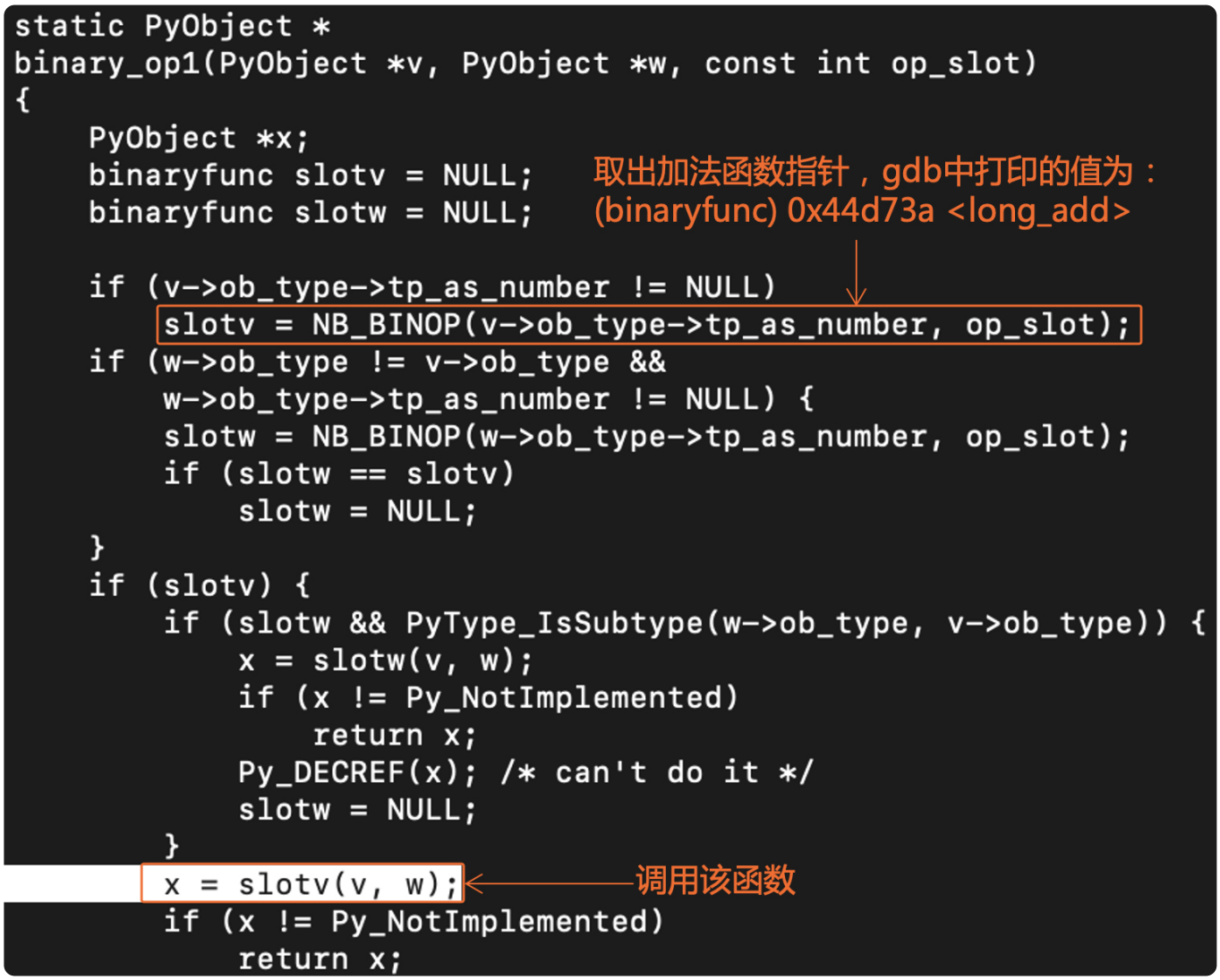
可以看到，如果 + 号两边是字符串，那么编译器就会执行字符串连接操作。否则，就作为数字相加。

我们继续跟踪进入 PyNumber_Add 函数。在这个函数中，Python 求出了加法函数指针在 PyNumberMethods 结构体中的偏移量，接着就进入了 binary_op1() 函数。

```
PyObject *
PyNumber_Add(PyObject *v, PyObject *w)
{
    PyObject *result = binary_op1(v, w, NB_SLOT(nb_add));
    if (result == Py_NotImplemented) {
        PySequenceMethods *m = v->ob_type->tp_as_sequence;
        Py_DECREF(result);
        if (m && m->sq_concat) {
            return (*m->sq_concat)(v, w);
        }
        result = binop_type_error(v, w, "+");
    }
    return result;
}
```

加法函数在 PyNumberMethods 结构体中的偏移量

在 binary_op1 函数中，Python 首先从第一个参数的类型对象中，取出了加法函数的指针。你在 GDB 中打印出输出信息，就会发现它是 binaryfunc 类型的，函数名称是 long_add。




```
static PyObject *
binary_op1(PyObject *v, PyObject *w, const int op_slot)
{
    PyObject *x;
    binaryfunc slotv = NULL;
    binaryfunc slotw = NULL;

    if (v->ob_type->tp_as_number != NULL)
        slotv = NB_BINOP(v->ob_type->tp_as_number, op_slot);
    if (w->ob_type != v->ob_type &&
        w->ob_type->tp_as_number != NULL) {
        slotw = NB_BINOP(w->ob_type->tp_as_number, op_slot);
        if (slotw == slotv)
            slotw = NULL;
    }
    if (slotv) {
        if (slotw && PyType_IsSubtype(w->ob_type, v->ob_type)) {
            x = slotw(v, w);
            if (x != Py_NotImplemented)
                return x;
            Py_DECREF(x); /* can't do it */
            slotw = NULL;
        }
        x = slotv(v, w);
        if (x != Py_NotImplemented)
            return x;
```

binaryfunc 类型的定义是：

复制代码

```
1 typedef PyObject * (*binaryfunc)(PyObject *, PyObject *);
```

也就是说，它是指向的函数要求有两个 Python 对象（的指针）作为参数，返回值也是一个 Python 对象（的指针）。

你再继续跟踪下去，会发现程序就进入到了 long_add 函数。这个函数是在 longobject.c 里定义的，是 Python 整型类型做加法计算的内置函数。

这里有一个隐秘的问题，**为什么是使用了第一个参数（也就是加法左边的对象）所关联的加法函数，而不是第二个参数的加法函数？**

在我们的示例程序中，由于加法两边的对象的类型是相同的，都是整型，所以它们所关联的加法函数是同一个。但是，如果两边的类型不一样怎么办呢？这个其实是一个很有意思的函数分派问题，你可以先思考一下答案，我会在后面讲 Julia 的编译器时再回到这个问题上。

好了，现在我们就能理解了，像加减乘除这样运算，它们在 Python 里都是怎么实现的了。Python 是到对象的类型中，去查找针对这些运算的函数来执行。

除了内置的函数，我们也可以自己写这样的函数，并被 Python 所调用。来看看下面的示例程序，我们定义了一个“__add__”魔术方法。这个方法会被 Python 作为 SimpleComplex 的加法函数所使用，实现了加法操作符的重载，从而支持复数的加法操作。

复制代码

```
class SimpleComplex(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "x: %d, y: %d" % (self.x, self.y)
    def __add__(self, other):
        return SimpleComplex(self.x + other.x, self.y + other.y)

a = SimpleComplex(1, 2)
b = SimpleComplex(3, 4)
c = a + b
print(c)
```

那么对于这么一个自定义类，在执行 BINARY_ADD 指令时会有什么不同呢？通过 GDB 做跟踪，你会发现几点不同：

首先，在 SimpleComplex 的 type 对象中，add 函数插槽里放了一个 slot_nb_add() 函数指针，这个函数会到对象里查找“__add__”函数。因为 Python 的一切都是对象，因此它找到的是一个函数对象。

所以，接下来，Python 需要运行这个函数对象，而不是用 C 语言写的内置函数。**那么怎么运行这个函数对象呢？**

这就需要用到 Python 的另一个协议，**Callable 协议**。这个协议规定，只要为对象的类型中的 tp_call 属性定义了一个合法的函数，那么该对象就是可被调用的。

对于自定义的函数，Python 会设置一个缺省的 tp_call 函数。这个函数所做的事情，实际上就是找到该函数所编译出来的 PyCodeObject，并让解释器执行其中的字节码！

好了，通过上面的示例程序，我们加深了对类型对象的了解，也了解了 Python 关于数值计算和可调用性（Callable）方面的协议。

Python 还有其他几个协议，比如枚举协议和映射协议等，用来支持对象的枚举、把对象加入字典等操作。你可以利用这些协议，充分融入到 Python 语言的框架中，比如说，你可以重载加减乘除等运算。

接下来，我们再运用 Callable 协议的知识，来探究一下 Python 对象的创建机制。

Python 对象的创建

用 Python 语言，我们可以编写 class，来支持自定义的类型。我们来看一段示例代码：

复制代码

```
class myclass:
    def __init__(self, x):
        self.x = x
    def foo(self, b):
        c = self.x + b
        return c
a = myclass(2);
```

其中，myclass(2) 是生成了一个 myclass 对象。

可是，你发现没有，Python 创建一个对象实例的方式，其实跟调用一个函数没啥区别（不像 Java 语言，还需要 new 关键字）。如果你不知道 myclass 是一个自定义的类，你会以为只是在调用一个函数而已。

不过，我们前面已经提到了 Python 的 Callable 协议。所以，利用这个协议，任何对象只要在其类型中定义了 tp_call，那么就都可以被调用。

我再举个例子，加深你对 Callable 协议的理解。在下面的示例程序中，我定义了一个类型 Bar，并创建了一个对象 b。

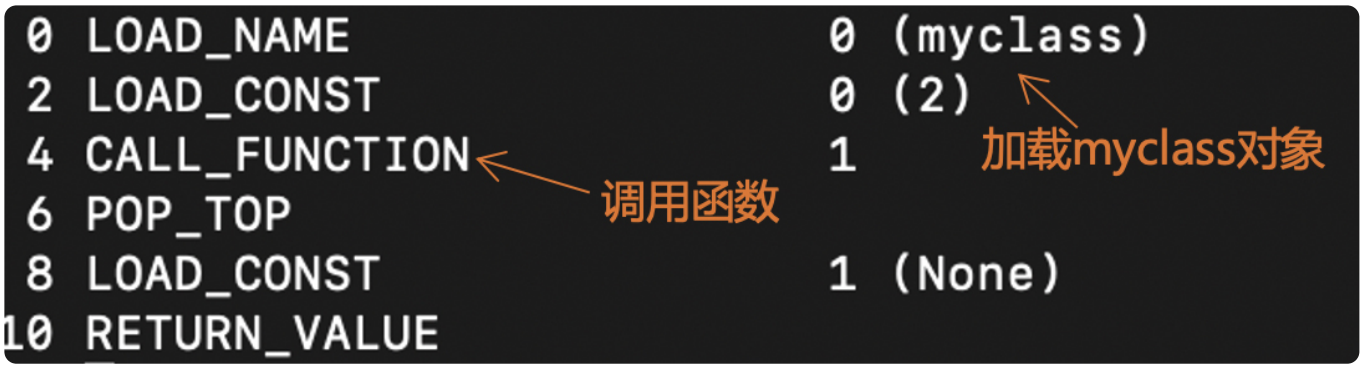
复制代码

```
class Bar:
    def __call__(self):
            print("in __call__: ", self)
b = Bar()
b()                                  #这里会打印对象信息，并显示对象地址
```

现在，我在 b 对象后面加一对括号，就可以调用 b 了！实际执行的就是 Bar 的“__call__”函数（缺省的 tp_call 函数会查找“__call__”属性，并调用）。

所以，我们调用 myclass()，那一定是因为 myclass 的类型对象中定义了 tp_call。

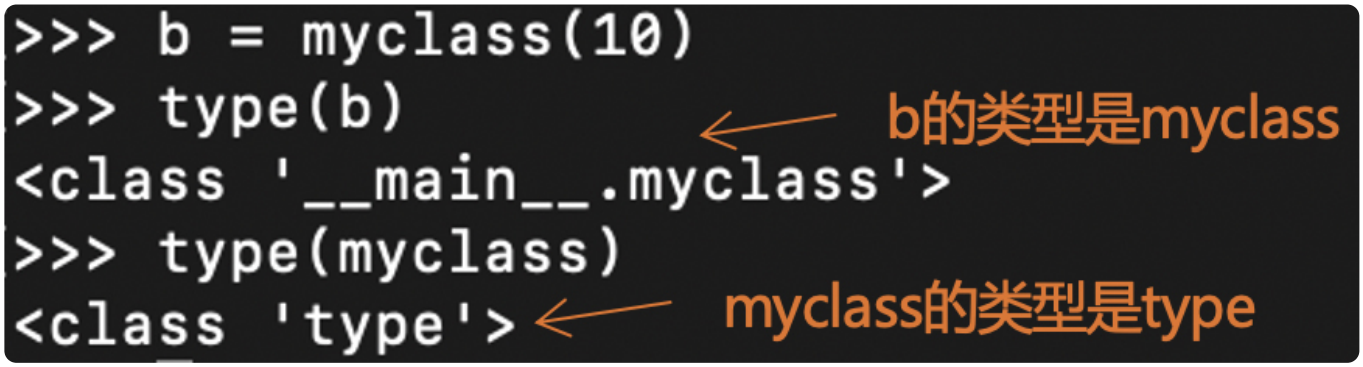
你还可以把“myclass(2)”这个语句编译成字节码看看，它生成的是 CALL_FUNCTION 指令，与函数调用没有任何区别。



可是，我们知道，示例程序中 a 的类型对象是 myclass，但 myclass 的类型对象是什么呢？

换句话说，一个普通的对象的类型，是一个类型对象。**那么一个类型对象的类型又是什么呢？**

答案是**元类（metaclass）**，元类是类型的类型。举例来说，整型的 metaclass 是 PyType_Type。其实，大部分类型的 metaclass 是 PyType_Type。



所以说，调用类型来实例化一个对象，就是调用 PyType_Type 的 tp_call 函数。**那么 PyType_Type 的 tp_call 函数都做了些什么事情呢？**

这个函数是 type_call()，它也是在 typeobject.c 中定义的。Python 以 type_call() 为入口，会完成创建一个对象的过程：

创建

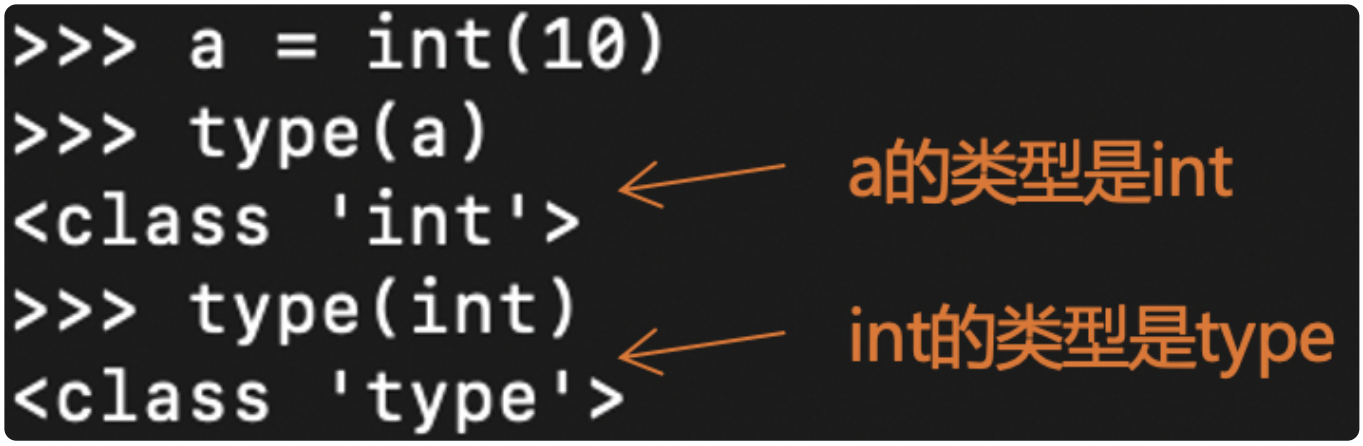
tp_call 会调用类型对象的 tp_new 插槽的函数。对于 PyLong_Type 来说，它是 long_new。

如果我们是创建一个 Point 对象，如果你为它定义了一个“__new__”函数，那么就将调用这个函数来创建对象，否则，就会查找基类中的 tp_new。

初始化

tp_call 会调用类型对象的 tp_init。对于 Point 这样的自定义类型来说，如果定义了“__init__”函数，就会执行来做初始化。否则，就会调用基类的 tp_init。对于 PyBaseType_Object 来说，这个函数是 object_init。

除了自定义的类型，内置类型的对象也可以用类型名称加括号的方式来创建。我还是以整型为例，创建一个整型对象，也可以用“int(10)”这种格式，其中 int 是类型名称。而且，它的 metaclass 也是 PyType_Type。



当然，你也可以给你的类型指定另一个 metaclass，从而支持不同的对象创建和初始化功能。虽然大部分情况下你不需要这么做，但这种可定制的能力，就为你编写某些特殊的功能（比如元编程）提供了可能性。

好了，现在你已经知道，类型的类型是元类（metaclass），它能为类型的调用提供支持。你可能进一步会问，**那么元类的类型又是什么呢？是否还有元元类？直接调用元类又会发生什么呢？**

缺省情况下，PyType_Type 的类型仍然是 PyType_Type，也就是指向它自身。对元类做调用，也一样会启动上面的 tp_call() 过程。

到目前为止，我们谈论 Python 中的对象，还没有谈论那些面向对象的传统话题：继承啦、多态啦，等等。这些特性在 Python 中的实现，仍然只是在类型对象里设置一些字段即可。你可以在 tp_base 里设定基类（父类）来支持继承，甚至在 tp_bases 中设置多个基类来支持多重继承。所有对象缺省的基类是 object，tp_base 指向的是一个 PyBaseObject_Type 对象。

复制代码

```
1 >>> int.__base__      #查看int类型的基类
2 <class 'object'>
```

到目前为止，我们已经对于对象的类型、元类，以及对象之间的继承关系有了比较全面的了解，为了方便你重点复习和回顾，我把它们画成了一张图。

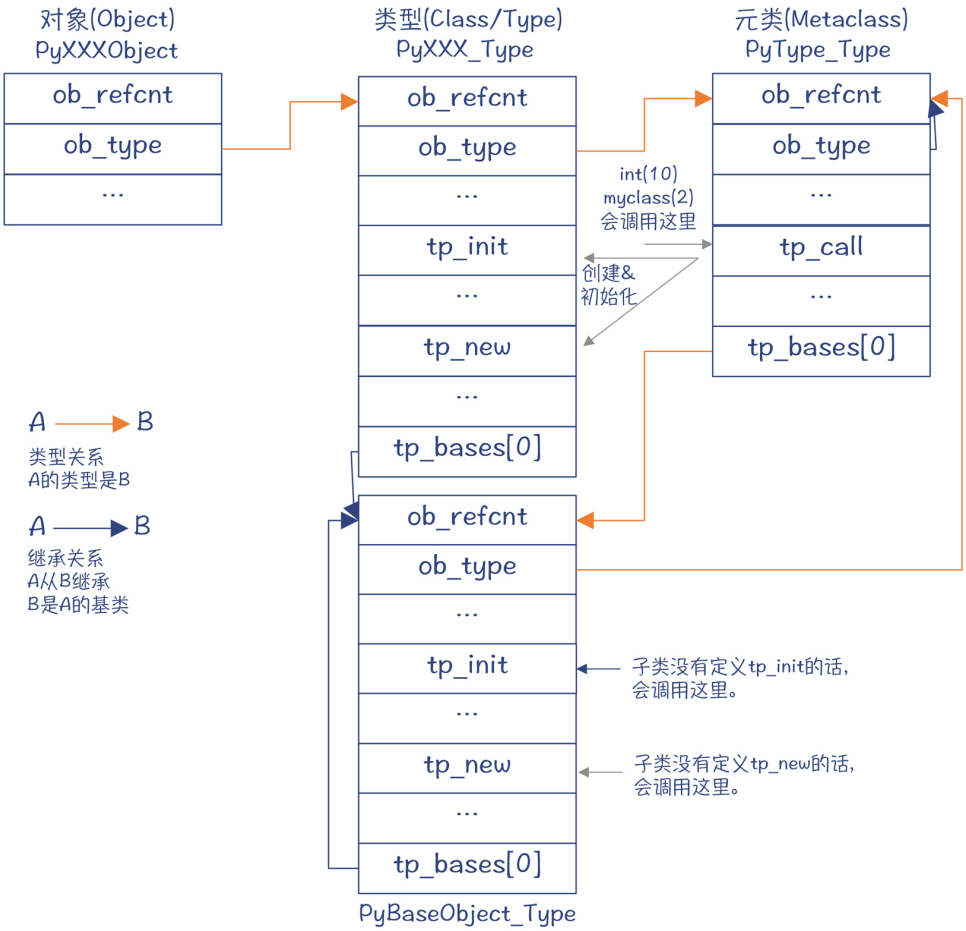


图 2：Python 对象的类型关系和继承关系

你要注意，图中我用两种颜色的箭头区分了两种关系。**一种是橙色箭头，代表的是类型关系**，比如 PyLong_Type 是 PyLongObject 的类型，而 PyType_Type 是 PyLong_Type 的类型；**另一种是黑色箭头，代表的是继承关系**，比如 int 的基类是 object，所以 PyLong_Type 的 tp_base 指向 PyBaseObject_Type。

到这里，你可能会觉得有点挑战认知。因为通常我们谈面向对象的体系结构，只会涉及图中的继承关系线，不太会考虑其中的类型关系线。Python 的类型关系，体现了“数据即程序”的概念。Java 语言里，某个类型对应于一个 class 的字节码，而在 Python 里，一个类型只是一个 Python 对象而已。

并且，在 Java 里也不会有元类，因为对象的创建和初始化过程都是语言里规定死的。而在 Python 里，你却拥有一定的掌控能力。

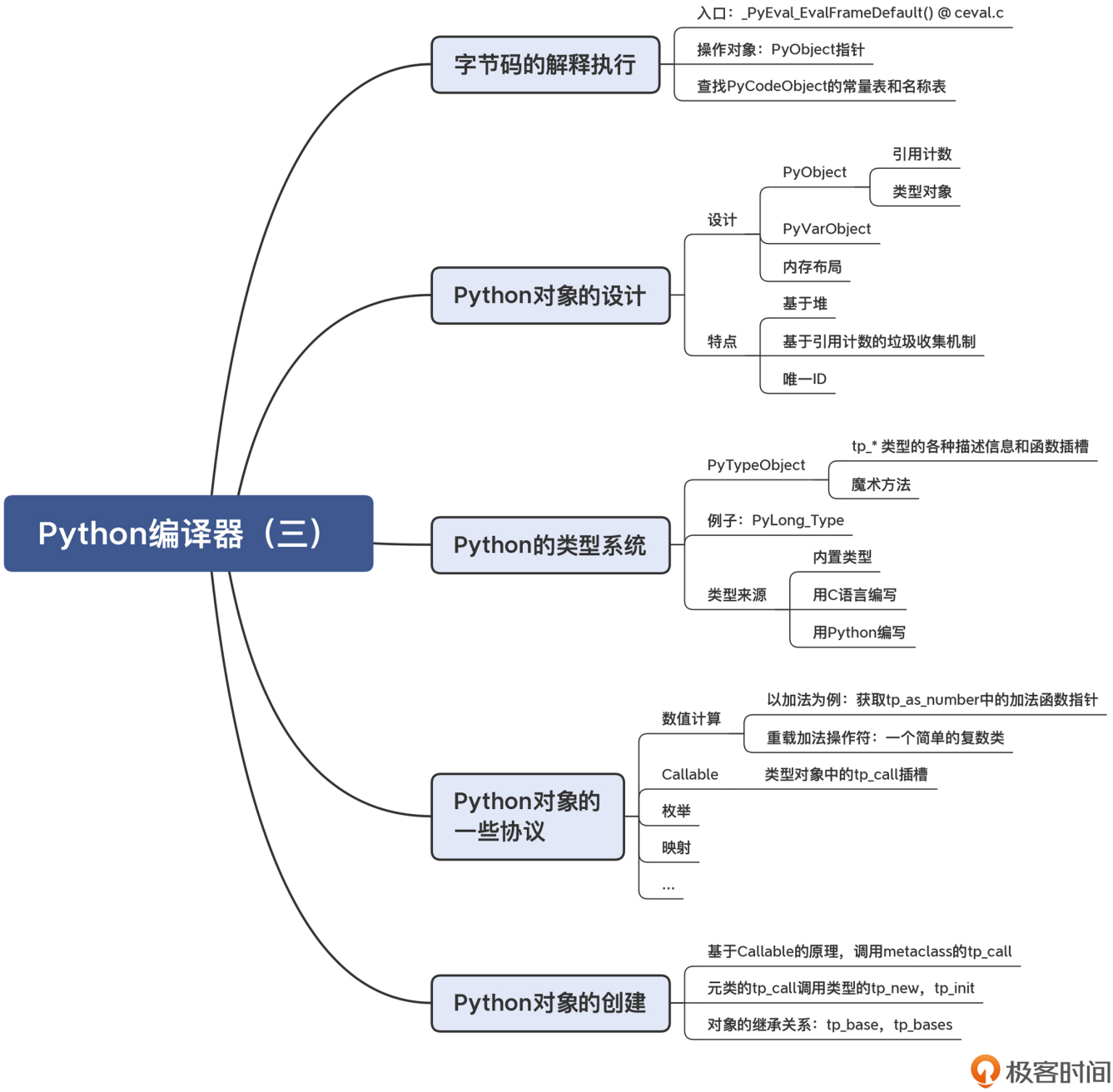
这些特点，都体现了 Python 类型体系的强大之处。

课程小结

好了，我们来总结一下 Python 运行期的特征。你会发现，Python 的运行期设计的核心，就是 **PyObject 对象**，Python 对象所有的特性都是从 PyObject 的设计中延伸出来的，给人一种精巧的美感。

> Python 程序中的符号都是 Python 对象，栈机中存的也都是 Python 对象指针。
>
> 所有对象的头部信息是相同的，而后面的信息可扩展。这就让 Python 可以用 PyObject 指针来访问各种对象，这种设计技巧你需要掌握。
>
> 每个对象都有类型，类型描述信息在一个类型对象里。系统内有内置的类型对象，你也可以通过 C 语言或 Python 语言创建新的类型对象，从而支持新的类型。
>
> 类型对象里有一些字段保存了一些函数指针，用于完成数值计算、比较等功能。这是 Python 指定的接口协议，符合这些协议的程序可以被无缝集成到 Python 语言的框架中，比如支持加减乘除运算。
>
> 函数的运行、对象的创建，都源于 Python 的 Callable 协议，也就是在类型对象中制定 tp_call 函数。面向对象的特性，也是通过在类型对象里建立与基类的链接而实现的。

我照例把本讲的重点知识，整理成了一张思维导图，供你参考和回顾：



一课一思

今天给你的思考题是很有意思的。

我前面讲到，当 Python 做加法运算的时候，如果对象类型相同，那么只有一个加法函数可选。但如果两边的对象类型是不同的，该怎么办呢？你可以看看 Python 是怎么实现的。这其实是编译技术的一个关注点，我们在后面课程中还会提及这个问题。

参考资料

Python 的内置类型。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | Python编译器（二）：从AST到字节码

下一篇 20 | JavaScript编译器（一）：V8的解析和编译过程

精选留言（1）

写留言

飞翔

2020-07-17

python编译过程中会调到python的代码，pgen用来生成c语言的代码，那这个原始的Python编译器是用什么实现的？也就是第一个的Python编译器是怎么出来的？